Robotics Research Technical Report

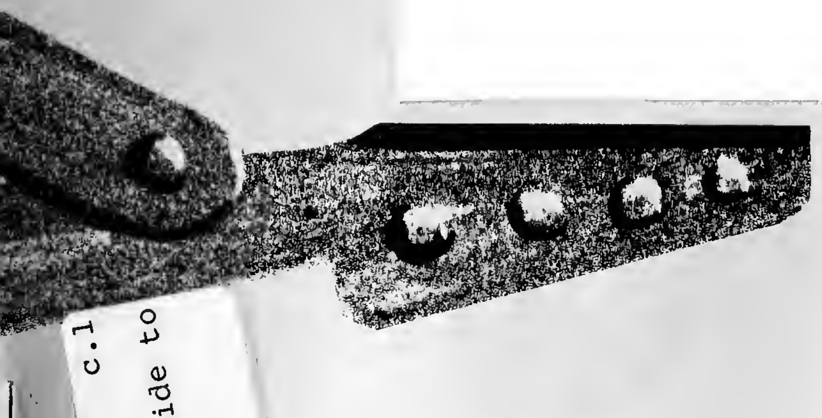
Generatorium omnis laboris ex machina

An Experimenter's Guide to the
Four-Finger Manipulator

by

Jim Fehlinger

Technical Report No. 282
Robotics Report No. 102
February, 1987



New York University
C Institute of Mathematical Sciences

Computer Science Division
251 Mercer Street New York, N.Y. 10012

An Experimenter's Guide to the
Four-Finger Manipulator

by

Jim Fehlinger

Technical Report No. 282
Robotics Report No. 102
February, 1987

New York University
Dept. of Computer Science
Courant Institute of Mathematical Sciences
251 Mercer Street
New York, New York 10012

Work on this paper has been supported by Office of Naval Research Grant N00014-82-K-0381, National Science Foundation CER Grant DCR-83-20085, and by grants from the Digital Equipment Corporation and the IBM Corporation.

Contents

1 Using the four-finger manipulator

The four-finger manipulator (known as *dexter*) is controlled by a Motorola 68000 processor which has a console terminal and connections via serial line and Ethernet to a host computer (currently the Sun workstation called *robroy*). The routines which operate dexter are invoked by typing commands on dexter's console, which are interpreted by a command shell called *conch*. Dexter's control program is created on robroy or one of the other Suns by linking the routines comprising conch together with other routines into a single executable file, which must be downloaded to dexter in order to run.

The four-finger control software is supported by a real-time, UNIX-based operating system kernel called *NRTX*. When dexter is used, both NRTX and the control program must be downloaded to dexter's controller by typing commands on robroy. To run an experiment or demonstration on dexter, the user should do the following:

Login to robroy, most conveniently on robroy itself (the Sun workstation nearest dexter).

Start the program on robroy which permits dexter's control program to read and write files on robroy via the Ethernet. This program is located in the directory */usr/local/nrtx*, which it is convenient to have in your search path. Assuming this directory will be searched, the appropriate command to robroy is '*fileserver &*'. Note that this fileserver should run in the background on robroy.

Check that dexter's motors are switched on. There are two strips of electrical outlets beneath dexter's table on the side opposite the controller electronics into which the motors are plugged. The illuminated rocker switch on each power strip should be on.

Check that dexter's controller is switched on. The controller is located beneath dexter's table, on the side to the left of robroy, in an equipment rack supported by castors. Both the power switch (an illuminated rocker switch) and the CPU reset switch (a square push button) are located in a black faceplate at the front and bottom of the equipment rack.

Press dexter's CPU reset switch, located in the black faceplate of the controller's equipment rack.

Observe dexter's console terminal, mounted on the swiveling terminal stand attached to robroy's desk. In response to CPU reset, the controller's PROM monitor will display some lines of text on the console followed by a command prompt '>' in the left margin. If this prompt fails to appear, press the CPU reset switch again.

Type '*ib*' on dexter's console when the PROM monitor command prompt appears. Dexter is now ready to receive downloaded software from robroy.

Start NRTX on dexter by typing the command '*nrtx dexter*' on robroy (again assuming that the directory */usr/local/nrtx* is in your search path). This command is actually a shell script which performs three distinct functions in succession: (1) It downloads a bootstrap program via the serial connection between robroy and dexter. (2) It boots NRTX via the Ethernet. (3) It downloads a program which initializes the Sky floating-point processor board in dexter's controller. During (1), approximately one line of characters will appear on both robroy's screen (lower-case *y*'s) and dexter's console screen (a repeated sequence of ten characters). When (2) is in progress, the message '*Ethernet boot*' appears at the beginning of a line on dexter's screen, followed by a line of (mostly) *y*'s, followed by '*Done*'. If (2) fails, the *y*'s will stop appearing on dexter's screen, and the message '*etherboot: boot failure*' will appear on robroy's screen. It is then necessary to reset dexter's CPU and start over. During (3), several messages will appear on dexter's screen, the last one of which should be '*EXIT STATUS 0*'.

Start the four-finger manipulator control program on dexter by typing the command '*pl -hdexter /usr/src/local/ffm/bin/conch*' on robroy (still assuming that the directory */usr/local/nrtx* will be searched). Note that the final argument in the above command may be any executable file prepared for dexter using the four-finger development tools on the Suns.

When it begins executing, dexter's control program will ask the user (on dexter's console screen) if the host fileserver will be needed. In general, the answer to this question will be '*y*'. The further question is then asked whether the fileserver is running on the host. After making sure this is the case, the user should also respond '*y*' to this question. When the connection to robroy is successfully established, conch executes a file of commands on robroy named */usr/src/local/ffm/.conchrc.*

If dexter cannot make a connection with robroy, it will display the message '*Connection to robroy failed*' followed by a conch command prompt. At this point the user should kill any fileservers running on robroy, start a fresh fileserver, and type the command '*init*' on dexter. Dexter will ask again whether the fileserver is running on robroy, to which the response is '*y*'. If the connection is successful this time (as it should be), the .conchrc file will then be executed by dexter.

One of the commands in the .conchrc file which dexter executes upon startup is the command '*mcfindhome*', which returns the fingers to their home positions (with each motor's trolley in its outermost position). This command, before moving the fingers, asks the user if it is all right to send the fingers home. Before replying '*y*', the user should make sure that no objects block any finger's path on dexter's table.

When the commands in the .conchrc file have been executed, conch prints a command prompt on dexter's screen and awaits further commands from the keyboard. Typing '?' or '*help*' on dexter's console causes a list of all the available

conch commands to be displayed. Typing a command name as an argument to the '*help*' command causes the command named by the argument to be described briefly. It is possible for the experimenter to add commands to the control program, as described below in the section entitled *The four-finger development environment*.

2 Overview of the four-finger software

The NRTX operating system which supports the control software for the four-finger manipulator (*dexter*) permits multiple concurrent processes to run on dexter. Each such process shares its code and static data address spaces with all other processes, but has its own private stack. Dexter's control software makes use of this feature of NRTX to divide its activities into a number of asynchronous processes. This multiprocessing structure of dexter's control software should allow it to make good use of a multiprocessor-based controller (such as the Ultracomputer), if such a device were to be fitted to dexter in the future.

The first process to execute on dexter when the control program is downloaded from robroy is the conch command interpreter. This process, after some initialization code, reads a command which is typed on dexter's console keyboard (or which is listed in a batch command file on robroy), matches the command string with an entry in a table of commands, and calls the corresponding function given in the command table. The command-line arguments are made available to the function in the same way that a UNIX shell provides arguments to the main procedure of a program (e.g., by means of the *argc* and *argv*[ ] conventions in C).

As part of its initialization, before accepting commands from the keyboard, conch executes the commands listed in the file */usr/src/local/ffm/.conchrc* on robroy. These commands are echoed to dexter's console screen as they are read by conch. Afterward, any similar batch command file may be executed by typing the '*batch*' command on dexter. This command takes as its sole argument the name of a file on robroy containing the commands, listed one per line (any line starting with a '#' is a comment, and is echoed on dexter's screen but otherwise ignored). After the last command in a batch file is executed, conch resumes taking commands from dexter's keyboard. A batch command file may itself contain a '*batch*' command, but such a command is analogous to a goto statement and not a function call, since conch returns to the keyboard rather than the rest of the first command file when the second command file ends.

If the filename argument of a '*batch*' command begins with the character '/', the argument is interpreted as the absolute pathname of a file on robroy. Otherwise, the filename is constructed by prepending to the supplied argument a prefix string whose default value is '*/usr/src/local/ffm/*' (note the terminal '/'). The value of this prefix string may be changed by giving the desired string as argument to the '*path*' command. When issued without an argument, the '*path*' command displays on dexter's screen the current value of the prefix string for batch files on robroy.

During initialization, conch spawns a second process on dexter, the *timer* process. This process utilizes the NRTX *alarm*( ) system call to provide a clock signal regulating the gathering of data from dexter's motors and fingers. The timer maintains three clock intervals, the *basic interval*, the *cycle interval*, and

the *heartbeat interval.* The heartbeat interval is not currently used. The basic interval regulates how often a global time stamp is updated. The cycle interval regulates how often information is gathered from dexter's motors and fingers. The software forces the cycle interval to be greater than or equal to the basic interval, but in general they are kept the same.

The values of the timer's clock intervals are given in milliseconds, and are restricted by NRTX to multiples of 10. Default values of these intervals are given in the macros *BASIC_INTERVAL*, *CYCLE_INTERVAL*, and *HEARTBEAT_INTERVAL*, defined in the header file *parameters.h* in the directory */usr/src/local/ffm/nsrc/h* on the Sun network, but they may be altered dynamically on dexter by typing the '*tinit*' command. This command is generally given two identical arguments, the basic interval and the cycle interval. The .conchrc file currently contains the command '*tinit 100 100*'.

The '*mcinit*' command in .conchrc causes conch to spawn a third process on dexter, called *mcprocess.* Mcprocess, restarted after each cycle interval by the timer, reads each finger's strain gauges and each motor's motion status and position. If any strain-gauge reading exceeds a safe threshold, mcprocess will abort the experiment in progress and move the fingers so that the strain on each is reduced to zero.

Each type of reading for a particular finger or motor pair is recorded by mcprocess in the appropriate type of *frame.* A frame is a data structure allocated from a circular buffer when needed to record any event or periodically measured or calculated data of significance to dexter's control program. Pointers to frames allow the information recorded in them to be passed to any routine in the control program needing access to that information. In addition, selected frames may be sent across the Ethernet by a separate *archiver* process to a file on robroy for later examination and debugging of the experiment.

Each frame is preceded by a header giving its type and length and giving a time stamp and cycle number according to the clock intervals established by the timer process discussed above. There is a specific type of frame associated with each type of information (e.g., strain-gauge reading, finger position, or motor-controller command) used by the control program. Frames of the above-mentioned types requested by mcprocess after each cycle interval provide the most recent information about the fingers to the motion-planning routines of dexter's control program.

Dexter's control program permits flexible allocation of fingers among motion-control routines by means of finger *groups.* The conch '*group*' command allows one or more fingers, identified by the numbers 1, 2, 3, or 4, to be assigned to one or more groups, identified by the letters *a*, *b*, *c*, or *d*. Dexter's fingers are numbered clockwise starting from the side of the table above the controller electronics.

The '*group*' command takes as arguments a group-id followed by one or more finger-ids. For example, the command '*group a 1 2 3*' establishes a finger group

a containing fingers 1, 2, and 3. Other conch commands might now be used, with the group-id *a* as an argument, to grasp or move objects with that finger group.

If the finger-ids are omitted from the '*group*' command, conch interprets the command as a request for a list of the members of the specified finger group. For example, the command '*group a*' might result in the message '*Finger group 'a' contains: 1, 2, 3*'. A '*group*' command with no argument is a request for a list of the members of all four finger groups.

The '*ungroup*' command with a group-id argument causes conch to dissolve the specified finger group (i.e., to cause it to have no members). The same command with no arguments dissolves all four finger groups.

Once the fingers chosen to participate in an experiment are associated in finger groups, the '*ready*' command moves the fingers into appropriate preliminary positions for the experiment. A group-id may be given as the first argument to the command, causing the command to be applied to the specified finger group. If no group-id is given, the command applies to all non-empty finger groups. The next argument must be a name appropriate to the experiment being performed, which allows conch to determine the correct initial positions for the fingers chosen to take part in the experiment. For example, the command '*ready a disk*' will cause the fingers in group *a* (there should be three of them in this case) to extend toward the center of dexter's table in preparation for touching and grasping the ten-centimeter-diameter plastic disk.

Before the fingers are actually moved, conch's response to the '*ready*' command is to spawn a group-control process, if one does not already exist, for each finger group explicitly or implicitly specified in the command. From this point on, the original (master) conch continues fetching commands from a batch file or the keyboard. If one of these commands refers to one or more finger groups, either explicitly or implicitly, the master conch sends a command message to a subsidiary conch in the group-control process for each finger group referred to in the command being processed by the master conch. Thus, if more than one finger group is involved in an experiment, the motion of each group's set of fingers is determined by a separate concurrent group-control process.

It is important to be certain that no obstacle is in the path of any finger before typing the '*ready*' command or any other command which causes the fingers to move rapidly. Such commands include the '*mcfindhome*' command listed in the .conchrc file, the similar '*mcready*' command which sends the fingers home, and the low-level conch commands that control finger motion directly ('*mcamove*', '*mcdmove*', etc.). These result in finger motion which is too fast for mcprocess to halt in time to prevent damage if safe strain-gauge limits are exceeded.

In contrast, motion-control commands whose actions are mediated by a finger group result in finger movement which occurs in such small steps that mcprocess acts as an effective safeguard against damage. These commands (such as

'*translate*' and '*rotate*') are characterized by the fact that they may take a group-id as a first argument. However, the most important safety feature at all times is a cautious and alert operator.

To continue with the example of an experiment utilizing dexter to grasp and move a ten-centimeter plastic disk, assume that we have a single group *a* containing fingers 1, 2, and 3, and that the fingers have been moved to their preliminary positions with the '*ready*' command described above. The disk should be in the center of dexter's table, supported near its center by one of the large metal nuts provided for this purpose. The three fingers, in their initial positions for this experiment, will surround the disk, each at a distance from it of two centimeters or less.

The next command that should be typed on dexter's console is '*touch a*'. This command will cause the fingers in group *a* to make contact with the disk. Since the fingers do not currently move into contact simultaneously, it is necessary for the experimenter, or an assistant, to hold the disk down at its center in order to prevent it from being knocked off its supporting nut until at least two fingers have made contact.

As with the '*ready*' command, any conch command that expects a group-id as its first argument will apply the command only to the specified group if the group-id is supplied, or to all non-empty groups if the group-id is omitted. Thus, since only one finger group is being used in our example experiment, the above command would normally be shortened to '*touch*'.

The next command to dexter in the experiment under consideration is '*initialize a*' (or simply '*initialize*', since we are only dealing with one finger group). This command initializes three data structures for each finger group specified, in accordance with the experiment in progress: the *parameters*, the *coefficients*, and the *initial values*. These data structures are declared in the header file */usr/src/local/ffm/nsrc/h/control.h* on the Sun network, where they are given the defined types *PARMS*, *COEFFS*, and *VALUES*, respectively.

The fields of a PARMS structure contain experimental parameters such as the minimum force to be applied to each finger when gripping a body, or the maximum distance a finger may move in a single step. The COEFFS structure contains computational data such as matrices used to compute finger gripping positions and forces given a desired position and orientation of the gripped body (see the theoretical discussions by Schwartz and Maw-Kae). The VALUES structure contains data on the current state of the experiment, such as the top and bottom positions of a group's fingers, the force on each finger, the position and orientation of the gripped body, and the force and torque on the gripped body. Each of these data structures is also copied into a frame of the appropriate type, in order that it may be archived to robroy if desired.

A number of conch commands may now be used in our example experiment to cause dexter to control the motion of the plastic disk with the finger group we have established. Each of these commands takes a group-id as its first

argument, which may be omitted in our example. Some of these commands take additional arguments, which may be optional. Among the motion-control commands currently available for use with the plastic disk are '*grasp*', '*follow*', '*translate*', and '*rotate*'.

The '*grasp*' command causes dexter to grip the disk with enough force to prevent it from slipping, and to oppose any external force applied to the disk. This command should be used before typing any of the commands which cause the disk to move with respect to the table, because while '*grasp*' is in effect the metal nut supporting the disk may be removed by poking it out from underneath with some other object (a tape write-protect ring is convenient for this purpose). Removing this nut prevents any friction it might introduce from fouling the further course of the experiment.

All of the motion-control commands listed above are carried out by means of an *inner-control routine* which utilizes an appropriate *planner*. The inner-control routine for a particular finger group is executed repeatedly by that group's control process as a result of the command message sent to it by the master conch, which has in turn responded to the motion-control command typed on dexter's keyboard. Each time it is called, the inner-control routine is given a pointer to a planner, a motion-planning routine appropriate to the motion-control command which that group is expected to carry out.

These *inner-control cycles* should not be confused with the mcprocess cycles. The latter occur at a rate determined by the timer process, and govern the gathering of raw data from the motors and fingers. The former occur at a rate limited by the computational speed of dexter's CPU and by the fact that finger motion is required to begin and end for each inner-control cycle before the next one begins. Currently, the repetition rate of the mcprocess cycles is about ten times that of the inner-control cycles, but they are not precisely synchronized.

For each inner-control cycle, two frames are allocated, each containing a VALUES structure. The *actual values* structure contains the state of dexter at the start of the inner-control cycle. This data is computed from information originating in the frames provided by mcprocess, containing strain-gauge and motor measurements, together with information in the COEFFS structure. The actual values, together with the *target values* structure (the second VALUES structure allocated by the inner-control routine), are passed to the planner, which computes target values corresponding to the target state which dexter must assume in order to carry out the desired motion. Finally, the inner-control routine uses the target values, in conjunction with the COEFFS, to compute *finger deltas*, changes in finger-top positions which are checked against maximum safe values, reduced (or *clipped*) if necessary, and then sent as commands to dexter's motors.

When a motion-control command such as '*grasp*', '*translate*' or '*rotate*' has been typed, the group control process will cycle through the inner-control routine a number of times determined by the planner routine *plangrasp*( ),

plantranslate( ) or *planrotate*( ), respectively, from arguments passed to it from the command line. In the case of motion commands such as '*follow*', the corresponding planner makes no such determination, and the motion will continue for a number of inner-control cycles given by the field *maximum_cycle* in the PARMS structure. In either case, a group's control process checks for new command messages in between calls to the inner-control routine, so that a new motion-control command typed on dexter's keyboard for a group currently executing a different (or the same) motion command should result in a smooth transition from the first motion to the second.

The '*stop*' command which, like the motion-control commands, takes a group-id as its first argument, will result in the cessation of inner-control cycles for the previously-typed motion-control command, if that command is still active. More drastically, a 'control-C' typed on dexter's keyboard causes the master conch's process to kill all group-control processes running on dexter and to try to establish zero-force readings on all fingers. To recover from this state, the '*ungroup*' command must be typed, then '*mcready*' to send the fingers home (check that the coast is clear for them), and then the experiment must be started over again with the sequence of commands described above.

In dire emergencies, the red panic buttons distributed around dexter's table stop all finger motion at the hardware level. If dexter's control program still seems to be functioning after such an event, it should be possible to recover by pressing the panic switch a second time (so that the light goes off) and then using the command sequence described above for 'control-C' recovery. Otherwise, the user must reboot dexter's controller and download NRTX and the control program from robroy again (remember to kill any fileserver still running on robroy and start a fresh one if dexter's control program terminates abnormally for any reason).

The '*follow*' command is similar to '*grasp*' in that the fingers are made to grip the disk tightly enough to prevent slippage. However, instead of resisting an external force applied to the disk, in this case the fingers move in the direction of the external force, permitting the disk to be pushed around in a horizontal plane.

The '*translate*' command moves the *tip* of the grasped body (defined as the center in the case of the plastic disk) to a point specified by x and y displacements in centimeters from the current tip position. The positive x and y directions are defined as to the right and forward, respectively, when facing the table from the side of finger 1.

The '*translate*' command takes three arguments, the first two of which are obligatory while the third is optional. The first two arguments are the target x and y displacements, in centimeters, of the grasped object's tip. The third argument is the distance the grasped body must travel during a single inner-control cycle, in centimeters. If omitted, the displacement per cycle defaults to

0.001 cm. To avoid clipping of finger deltas, this argument should be no more than 0.02 cm.

The '*rotate*' command rotates the *long direction* of the grasped body (initially defined as the positive x direction from the center in the case of the disk) through an angle given in radians. A positive angle indicates counterclockwise rotation. The angle is specified by the first argument to '*rotate*', while the optional second argument gives the amount of rotation to be accomplished during a single inner-control cycle. The rotation per cycle defaults to 0.004 radians if no second argument is given, and should be no more than 0.02 radians to avoid clipping finger deltas.

3 Using the archiver

The sequentially-allocated data structures referred to as *frames* permit periodically measured or computed information of significance to dexter's control software to be recorded in a central place for reference by any routine in the control program that needs it. The stream of frames may also be copied across the Ethernet to robroy for preservation in a file, which allows detailed data from an experiment on dexter to be examined off-line at the experimenter's convenience and with the help of statistical analysis tools on the Suns.

The information gathered into frames ranges from the very lowest-level control and status data for the strain gauges and motors, through intermediate-level finger motion requests, finger-force data, and finger-position data, to the highest-level actual-values and target-values frames allocated for each inner-control cycle. Additionally, all commands processed by the master conch or the subsidiary conchs in the group-control processes are recorded in command frames, while various events (such as the initiation or termination of processes, the opening and closing of fingers or motors, and error conditions) are recorded in event frames.

Each awakening of mcprocess after a cycle interval is recorded in a cycle frame, as well as being reflected in the cycle sequence number attached to each frame. Do not confuse the mcprocess cycles marked by cycle frames and cycle sequence numbers with the inner-control cycles, which are indicated by the appearance of actual-values and target-values frames in the frame stream.

Since it is generally not necessary to save all of this information for later examination, the user may select a subset of frame types to be included in the stream of data being copied to robroy. This is possible because each frame type is allocated from one of two circular buffers, the *main frame pool* or the *alternate frame pool*, but only frames from the main frame pool will be sent to robroy when the archiver process is running. The '*archive*' and '*noarchive*' commands cause frame types to be allocated from the main frame pool or the alternate frame pool, respectively.

The '*archive*' command takes as arguments one or more names of the frame types which are to be allocated from the main frame pool. If the command is given with no arguments, a list of the frame types currently being allocated from the main frame pool will be displayed. The '*noarchive*' command takes exactly the same arguments as the '*archive*' command, but in this case the named frame types are caused to be allocated from the alternate frame pool, or else a list of frame types currently being allocated from the alternate frame pool is displayed. Both commands may also take a wildcard '*' as argument, indicating all frame types, so that for example '*noarchive* *' would cause no frames to be sent to robroy even with the archiver process active.

The names of the frame types as they should appear in the argument list of an '*archive*' or '*noarchive*' command are the same as they would appear in a list

of frame types displayed on dexter's screen in response to typing one of these commands with no arguments. The frame-type names are intended to have some mnemonic value, such as *fgfor* for the finger-force frame type, or *sgframe* for the strain-gauge-reading frame type. Each name is a lower-case version of the name of the frame's corresponding defined-type structure declared in the header files in the directory */usr/src/local/ffm/nsrc/h* on the Suns.

There are currently about twenty different frame types, all declared in the above-mentioned header files. The frame types *ctpar* (inner-control parameters), *ctcoe* (inner-control coefficients), *ctval* (inner-control values), *cacmd* (conch command), and *aropen* (archiver open, inserted into the frame stream when the archiver process is started and whenever a comment is placed into the frame stream for documentation purposes) are allocated by default from the main frame pool, and are thus copied to robroy by default when the archiver process is created. These are the frame types which are the most useful for debugging a new motion-control planner.

The rest of the frame types currently used by the four-finger software are: *fgamov* (finger absolute move), *fgdmov* (finger displacement move), *fgopen* (finger open), *fgpos* (finger position), *fgvmov* (finger vector move), *fmevent* (general event), *mccmd* (motor command), *mcctl* (motor recalibration), *mcreq* (motor status request), *mcspur* (motor spurious message), *mcvelo* (motor velocity stream), *sgctl* (strain-gauge control).

The archiver is started on dexter by typing the command '*arstart*'. This command causes the master conch to spawn an asynchronous process which periodically copies to robroy, via the Ethernet, all the frames in the main frame pool which have been allocated since the last time the archiver ran. When a block of frames is archived, the bytes occupied by these frames at the tail end of the contiguous block of currently-used storage in the main frame pool are released for future use as new frames.

The archiver is periodically awakened after a number of mcprocess cycles whose default value is given by the macro *CYCLES_TO_WAIT* in the header file */usr/src/local/ffm/nsrc/h/parameters.h*. This waiting period may be altered by typing the command '*arwait*'. When given an argument, the '*arwait*' command alters the archiver's waiting period to the value of its argument; with no argument, the command causes the current archiver waiting period to be displayed on dexter's screen.

The most-recently-allocated frames are held for a number of mcprocess cycles before they may be archived, to make sure that information has been recorded to them before they are sent to robroy. This number is given in the macro *CYCLES_TO_HOLD* in the header file */usr/src/local/ffm/nsrc/h/parameters.h*, and is not currently alterable by means of a conch command. If the relative rates of the mcprocess cycles and inner-control cycles diverge more than they do currently, it may become necessary to increase this parameter.

When the command '*arstart*' is typed, the master conch attempts to open a file on robroy which will contain the archived frames. If this remote system call fails, conch will display the message '*Can't start archiver*' on dexter's screen. If this happens, it is necessary to kill any fileserver running on robroy, start a fresh fileserver, and reboot dexter.

The archive file which is opened on robroy is created by default in the directory */usr/tmp*. The pathname of this directory may be changed before starting the archiver by typing the command '*arpath*' on dexter, giving as argument the pathname on robroy in which to place the archive file. The '*arpath*' command with no argument will display the currently-selected pathname for archive files. The name of the archive file created on robroy is constructed from the characters '*ar_*' followed by the date and time of the experiment.

The first frame in the archive file is of type *aropen*, and contains a field of 256 characters which may contain an archive-file title. This title defaults to the same date and time as those contained in the file name, but may be altered before the archiver is started by typing the '*artitle*' command on dexter, which composes all its arguments into a line of text for the title field. Additional *aropen* frames may be inserted into the frame stream at any time by typing the '*arcomment*' command on dexter, followed by arguments to be inserted as a comment into the title field.

Once the archiver process and its associated archive file have been created, conch will ignore any subsequent '*arstart*' commands as along as the archiver is active. The '*archive*' and '*noarchive*' commands may be used at any time, even while the archiver is active, to change the set of frame types being sent to the archive file. The command '*arstatus*' causes dexter to display a message telling whether the archiver is active or inactive.

The command '*arstop*' causes all frames waiting to be archived to be immediately flushed to the archive file, the archive file to be closed, and the archiver process to terminate. If an archive file is empty save for the initial *aropen* frame, that file will be deleted.

A number of tools exist on the Suns for examining an archive file that has been created during the course of an experiment on dexter. The first tool to use is the program *arcopy*, to be found in the directory */usr/local/nrtx*. This program is a filter which takes an archive file containing binary data as input, and produces as output a human-readable text file identifying each frame by its type, frame sequnce number, and (mcprocess) cycle sequence number, and giving the name and value of each field of a frame.

Arcopy takes a number of options allowing the user to select specified frame types for inclusion in or exclusion from the output, or to select for output only frames from a specified range of frame sequence numbers or cycle sequence numbers. The current default selection of frames for arcopy is *aropen*, *cacmd*, *ctpar*, *ctcoe*, and *ctval*, the same set of frame types that is the default for the archiver on dexter. Arcopy's options are fully documented in an on-line-manual

entry on the Suns, which can be seen by typing the command '*man arcopy*'. The output of arcopy goes to *stdout*, so that the usual UNIX output-redirection capabilities may be used.

There are two messages which arcopy might send to *stderr*, and which would appear on the Sun terminal screen during its operation. The first of these is the message '*WARNING - NON-MONOTONIC FRAME SEQ NUMBER*'. This message is an indication that arcopy has read a frame whose frame sequence number is not greater than that of the previous frame in the archive file. This will arise if dexter's control software is allocating frames at a very high rate from the main frame pool.

If the head of the block of used storage in the main frame pool overtakes its tail, new frames could overwrite older frames after the archiver has requested their transmission to robroy and released their storage, but before they are actually copied across the Ethernet to robroy. A warning from arcopy that the frame sequence numbers are not monotonically increasing is an indication that the size of both the main and the alternate frame pools needs to be increased (by changing the macros *FRAME_POOL_SIZE* and *ALT_FRAME_POOL_SIZE* in the header file */usr/src/local/ffm/nsrc/h/parameters.h* and recompiling the control software).

If the main frame pool were to overflow, dexter's control software would not halt. Instead, all frames waiting to be archived would be discarded, and a frame of type *fmevent* would be placed in the frame stream with the event code *EE_ARCHIVER_TRASHED*. The macros giving event codes are defined in the header file */usr/src/local/ffm/nsrc/h/eecodes.h*.

It is, however, possible for dexter to seriously misbehave if it happens that the data in a frame from either the main or the alternate frame pool is overwritten before a routine requiring the information in the destroyed frame has completed its task. This did happen when an alternate-frame-pool size of 4 kilobytes was being used and all frames were being allocated from this pool (i.e., the command '*noarchive* $*$' was in effect). The symptoms of this were random and potentially damaging finger movements possibly indicating that motor-controller command frames were corrupted before the commands were sent to the motors.

Currently, 1 megabyte is reserved for the main frame pool, and 16 kilobytes for the alternate frame pool, which seems to be adequate. Dexter currently has 3 megabytes of memory, with NRTX occupying about 80 kilobytes and the control software about 1.5 megabytes, leaving more than 1 megabyte to spare.

The second error message that might appear on the Sun terminal screen when arcopy is running is '*WARNING - MISSING FRAMES*'. This message is an indication that the Ethernet controller on robroy lost one or more Ethernet packets from dexter. Rather than risk the possibility that dexter's control program will hang due to the loss of an acknowledgement message from the fileserver on robroy, we have adopted the strategy of disabling the machinery

for detecting and retransmitting lost packets on both dexter and robroy while the archiver is in operation. This means that we simply accept the loss of a packet from the archiver.

When the missing frames message appears on *stderr* during the operation of arcopy, a similar message is placed on arcopy's *stdout*. With the current default set of archived frames, and at the current cycle rates of mcprocess and the inner-control routine, it is seldom the case that Ethernet packets from the archiver on dexter will be lost by the fileserver on robroy.

Another error that can arise during use of the archiver has as its symptom the appearance of frames with many spurious zero-valued fields in the output of arcopy. This occurs because the frames are being transmitted to robroy by the archiver before information is being recorded in them, and is an indication that the parameter *CYCLES_TO_HOLD* described above is too small. Note that the spurious frame values are zero only before the entire address space of the main frame pool has been used at least once. If the difference between the frequencies of the mcprocess cycle and the inner-control cycle becomes greater than it is now, *CYCLES_TO_HOLD* will have to increase.

4 Using the S data analysis package

The archived data generated by an experiment on dexter may be manipulated and plotted using a data analysis package available on the Suns called *S*, and described in the book *S: An Interactive Environment for Data Analysis and Graphics* by Richard A. Becker and John M. Chambers. A number of S macros have been written to manipulate data from the archiver. The following description of these macros assumes that the user knows how to start S and is familiar with the syntax of S commands and the concept of an S dataset. Please refer to Becker and Chambers for a general discussion of S.

Before using S, the binary archive file from dexter must be changed into a text file by using the *arcopy* program on the Suns, as described in the last section. The output of arcopy should be redirected into a file in the directory from which S will be used. The S macros described below are used to examine the frames containing the initial VALUES, PARMS, and COEFFS structures created by the '*initialize*' command on dexter, and the actual and target VALUES structures created during execution of the inner-control loop for a motion-control command. These are the frame types which are preserved by default by arcopy.

The first macro which must be used in order to analyze archived data from dexter is '*sortffm*', which takes four or more arguments. The first three arguments are (1) the name of the text file output by arcopy, (2) the name of the motion-control conch command whose data is to be examined (e.g., translate, rotate, etc.), and (3) the number of fingers involved in the experiment on dexter. The last argument is one or more of the characters '*p*', '*c*', '*i*', '*a*', or '*t*'. These characters select for analysis frames of type *ctpar* (PARMS structure) or *ctcoe* (COEFFS structure); or frame type *ctval* (VALUES structure), subtypes *initial*, *actual*, or *target*, respectively.

In its current implementation, sortffm expects in its input only one frame each of type *ctpar* or *ctcoe*, and only one frame of type *ctval* and subtype *initial*. It is therefore recommended that the archiver on dexter be stopped and restarted before each use of the '*initialize*' command subsequent to the first, in order to create multiple archived-data files on robroy satisfying the requirements of sortffm. If this is not done on dexter, then prior to the use of S the output file from arcopy will have to be split manually into smaller pieces, each containing at most one of the above-mentioned frames.

Sortffm first executes the shell script */usr/local/nrtx/cleanarc*, passing to this script the name of the specified input file (arcopy's output file), the name of the motion-control conch command whose data is to be isolated, and the names of the chosen frame types (as they were specified to sortffm). Cleanarc extracts the specified data from the input text file and creates an output file corresponding to each selected frame type, with a name of the form *Sffmactual*, *Sffmtarget*, etc. These output files contain data in a form that fits the requirements of S.

Next, for each frame type specified in its argument list, sortffm calls another macro – one of *crparm*, *crcoef*, *crinit*, *cractual*, or *crtarget*. Each of these macros takes as argument the number of fingers used in the experiment on dexter. Each macro reads one of the corresponding input files created by cleanarc (*Sffmparm*, *Sffmcoef*, etc.), and creates a group of S datasets. Each group of datasets corresponds to a frame type whose data is collected in one of the *Sffm* files. Each dataset in the group contains the values of one field of the corresponding frame type.

The names of the S datasets created by the *cr* macros are mnemonically related to the names of the corresponding variables in the code for the inner-control loop in dexter's control program. For example, the successive values of the actual VALUES structure field *actual*→*tip_loc*[*X*] in the inner-control loop, as archived during the course of a motion-control command (in frames of type *ctval* and subtype *actual*), will be contained in the vector dataset *a.tiplocx*. Note that the prefix of the dataset name corresponds to the frame type *actual* specified in the call to sortffm. Similarly, the matrix dataset *t.fgforx* will contain successive values of the target VALUES structure field *target*→*finger_force*[*X*][], with one column corresponding to each finger in the experiment. During the creation of these datasets, sortffm displays a line of the form '*Read n items*', where *n* is an integer, for each frame type selected.

The datasets resulting from a call to sortffm will overwrite any previously-created datasets of the same name. If more than one motion-control command is to be analyzed during a given session with S, it is convenient to supply an additional prefix for the dataset names. This is done by using the S command '*prefix*' before each call to sortffm. For example, the user could type

```
> prefix("trans.")
> ?sortffm(transdata,translate,3,a)
> prefix("rot.")
> ?sortffm(rotdata,rotate,3,t)
```

resulting in the creation of datasets with prefixed names such as *trans.a.tiplocx* and *rot.t.tiplocy*.

Information on the use of sortffm and its related macros can be obtained on-line by typing the S '*help*' command. Once the appropriate S datasets have been created by these macros from the data gathered by the archiver on dexter, the full repertoire of S commands is available to manipulate and graph this data.

5 The four-finger development environment

The source code for dexter's software (largely written in C, but with a few mathematical routines in Fortran 77 and a very few files in assembler) is located in subdirectories of the directory */usr/src/local/ffm/nsrc* on the Sun network. These subdirectories are called *archive*, *arcopy*, *conch*, *control*, *ddt*, *finger*, *frame*, *h*, *matrix*, and *procopy*. The source files in these directories are kept very short, with usually no more than one function or subroutine per file.

Code relating to the archiver process is found in the *archive* directory. For example, the main loop of the archiver process is the routine *arloop*( ) in the file *arloop.c*, while the routine *arxmit*( ) in *arxmit.c* contains the function call which actually sends data across the Ethernet to the archive file on robroy. These routines, and the files which contain them, are prefixed by the letters *ar*.

The *arcopy* directory contains code for the arcopy program which is used to view archive files on the Suns. This directory and the *procopy* directory are unlike the others listed above in that they do not contain part of dexter's control program. The files in this directory and the routines which they contain are prefixed with the letters *ac*.

The *conch* directory contains code for the conch command interpreter. For example, the routine for processing a command string is located in the file *conch.c*, while the list of conch commands and their corresponding subroutines is in *castandard_table.c*. In the case of routines corresponding to conch commands which the user is expected to type, both the routines and the files which contain them are prefixed with the letters *ca*. An example is the file *caarstart.c* containing the routine *caarstart*( ), which is called when the user types the command '*arstart*' on dexter.

Other files in the conch directory, prefixed with the letters *gr*, are concerned with the establishment and maintenance of group-control processes for finger groups, and with the execution of command messages by these processes. For example, code for the main loop of a group-control process, which waits for a command message from the master conch and calls a subsidiary conch for its execution, is located in the file *groupprocess.c*. The routines in the master conch which send a command message to the group-control process are found in the files *grcommand.c* and *grsend.c*.

Miscellaneous other routines are also located in the conch directory, such as routines to translate the coordinates of a point on dexter's table from dexter's world coordinate system to the coordinate system of a particular finger (and *vice versa*). Another miscellaneous file is the assembly-language source file *monitorcall.s*, which contains code to escape to the CPU's ROM monitor for debugging purposes. The code for the timer process is also kept in this directory (in *timer.c*).

The *control* directory contains routines associated with the execution of the inner-control cycles. The routine *inner_control*( ) in the file *inner.c* fills in the

fields of the actual-values structure passed to it, calls the supplied planner, and calculates the finger deltas called for by the target values filled in by the planner. Other routines, such as the code in *coeffs.c*, is concerned with the initial computation of the coefficient matrices used by the inner-control routine during subsequent inner-control cycles.

Code for an assembly-language-level debugger called *ddt* is contained in the *ddt* directory. When making a control program for dexter, ddt may be included or left out at the programmer's option. The facilities provided by ddt are similar to those provided by the UNIX machine-level debugger *adb*, and include setting breakpoints, obtaining stack traces, and examining or changing the contents of variables or registers. More information about ddt is given below in the section entitled *Debugging and performance monitoring.*

The routines located in the *finger* directory are concerned at the lowest level with the strain gauges (files prefixed *sg*), motor pairs (files prefixed *mc*), and at an intermediate level with the fingers (files prefixed *fg*). The code for the process which periodically samples the strain gauges and the motor controllers is in the file *mcprocess.c*. An assembly-language file in this directory, *mcintr_glue.s*, is part of the handler of hardware interrupts from the motor controllers.

Routines which are concerned with managing the circular buffers from which the frames are allocated are kept in the *frame* directory, and are prefixed with the letters *fm*. The routine *fmcycle*( ) is called by the timer after each cycle interval, and this routine notifies both mcprocess (after each cycle) and the archiver (after *CYCLES_TO_WAIT* cycles) to perform their functions. *fmcycle*( ) also maintains a circular list of pointers to frames, called the *cycle list*, which facilitates the attribution of frames to cycles by the archiver.

The *h* directory contains the header files included by other four-finger source files. These contain mainly system-wide parameters and data structure declarations for the frames and other structures associated with the motors, fingers, and finger groups. There is some order dependency in the inclusion of these files. For example, *control.h* must be preceded by *finger.h*, which must be preceded by *motor.h*, which must be preceded by *parameters.h*. The last-mentioned file contains the most general system-wide parameters, and is included in most of the four-finger source files.

The *matrix* directory contains both C and Fortran files concerned with arithmetic such as matrix multiplication and eigenvalue calculation.

Finally, the *procopy* directory contains code for the procopy program which is used to view profiled-data files on the Suns. The files in this directory and the routines which they contain are prefixed with the letters *pro*. More information about profiling dexter's control program is given in the section entitled *Debugging and performance monitoring.*

The most likely modifications which the user of the four-finger manipulator is apt to wish to make to the software are (1) the addition of a new motion-

planning routine, (2) the addition of a new conch command, probably in association with (1) above, and (3) the enhancement of the archiver's debugging capability through the addition of a new type of frame. These additions can be be accomplished by imitating the structure of the existing code, and by using the UNIX *make* utility, which simplifies the process of compiling and linking a new control program.

To understand what is involved in adding a new planner to dexter's control program, consider what happens when a motion-control conch command, such as '*translate*', is typed. First, the command-table entry for '*translate*' in the master conch leads to the routine *grcommand*( ). This routine, with help from the routine *grsend*( ), changes the command to '*grtranslate*'. The modified command is sent as a message, together with any original command arguments (but *sans* any group-id), to the group-control processes implied or given by the first argument to '*translate*'.

The routine *groupprocess*( ) in the recipient group-control process receives the message and calls *conch*( ) with the received command as a parameter. A second reference to the command table maps the command string '*grtranslate*' to the routine *grtranslate*( ). The file *grtranslate.c* contains both the routine *grtranslate*( ) and the routine *plantranslate*( ), the latter being the actual motion planner for this command.

Finally, the routine *grtranslate*( ) calls the routine *grcontrol*( ), passing a pointer to the planner *plantranslate*( ) together with the command arguments. *grcontrol*( ) is a routine shared by all the motion-control commands which, for the correct number of inner-control cycles, allocates the actual-values and target-values frames and calls *inner_control*( ), giving it pointers to the PARMS, COEFFS, and actual and target VALUES structures, the planner, and the command arguments.

Suppose we wished to add the command '*gyrate*' to dexter's control program, together with a corresponding motion planner. A simple procedure would be simply to copy an existing motion-planner file, say *grtranslate.c* (in the *conch* directory on the Suns), to a new file *grgyrate.c*. Next, we edit the new file, changing occurrences of the string *translate* to *gyrate* (clean up the comments after this). Then, in the body of the routine *plangyrate*( ) in our new file, we substitute for the existing code new code that will calculate appropriate target values for the inner-control cycles in order to accomplish the new motion. If arguments are required by the new planner, they are supplied by exactly the same variables *plan_argc* and *plan_argv*[] as were used by *plantranslate*( ) (unlike the UNIX convention, the first argument past the group-id will be contained in *argv*[0]).

We must also edit the file *castandard_table.c*, adding an *extern* declaration for the new routine *grgyrate*( ), and adding two lines to the table *standardcaproc*[ ] – one associating the command '*gyrate*' with the routine *grcommand*( ), and one associating the command '*grgyrate*' with the routine *grgyrate*( ). Last, we

edit the file *Makefile*, which contains the UNIX *make* directives allowing us to rebuild dexter's control program. In *Makefile* we must add *grgyrate.c* to the list of *.c* files, and *grgyrate.o* to the list of *.o* files.

The command '*make install*' (we are still in the *conch* directory on the Suns) will now perform the steps necessary to construct a new control program for dexter. It will recompile the file *castandard_table.c*, which we altered, as well as compiling our new file *grgyrate.c*. It will rebuild the object archive *libconch.a* from the *.o* files in the *conch* directory, and copy this archive into the directory */usr/src/local/ffm/lib* (there is a corresponding object archive for each subdirectory of */usr/src/local/ffm/nsrc* containing source code for dexter's control program).

Finally, the *make* script will link together a new relocatable executable file from *main.o* in the *conch* directory, containing the program's entry point *main*( ), and the libraries *libarchive.a*, *libconch.a*, *libcontrol.a*, etc., in */usr/src/local/ffm/lib*. This new executable, called *conch*, is copied into the *bin* subdirectory of */usr/src/local/ffm*. Conch is still relocatable because robroy gets its final load address from dexter across the Ethernet; final relocation is part of the process of downloading the program.

Each of the four-finger development directories contains a *make* script in a file called *Makefile*. The compiler which is used by all of these scripts is */usr/local/nrtx/rtxcc*. It is actually a cross-compiler, generating Motorola 68000 code for dexter's CPU, whereas the standard Sun C compiler generates Motorola 68020 code. The NRTX include files used by some four-finger source files are located in the directory */usr/include/nrtx*.

The procedure for adding a conch command not associated with a finger group's motion planner is only slightly different from that described above. Suppose we wish to add the command '*date*', which will cause the date and time to be displayed on dexter's screen. This is a behavior not involved in any way with finger groups, and not requiring any message to be sent from the master conch to any group-control process.

Once again, we must edit the file *castandard_table.c*, but this time we add just one line instead of two to the table *standardcaproc*[ ] – an entry associating the command string '*date*' with the routine *cadate*( ). It is the established convention in the case of simple conch commands to form the routine name by prefixing the letters *ca* to the command name.

Next we create the file *cadate.c* containing the routine *cadate*( ). It is, once again, nomenclatural convention to make the name of the *.c* file correspond in this way to the name of the routine which it contains. Note that any command-line arguments that we might require are obtained by declaring the parameters *argc* and *argv*[ ] in the routine associated with the command name. These are declared exactly as are the standard parameters of *main*( ) in any C program, and serve exactly the same purpose.

We add *cadate.c* and *cadate.o* to *Makefile* in the *conch* directory as described above. Finally, we type '*make install*' to recreate the archive *libconch.a* and the relocatable executable file *conch*. Dexter will now have the new command in its repertoire.

It is possible to create a private version of dexter's control program, having a set of private commands which are declared in a separate file but which are merged by the software with the commands declared in *castandard_table.c*. To do this, copy the files *Makefile* and *main.c* from the *conch* directory into your own directory. Copy the file *castandard_table.c* from the *conch* directory to a file called *commands.c* in your directory. You will also, of course, need to have created the *.c* files containing the routines corresponding to the private commands you wish to implement.

Edit the file *commands.c*, changing the *extern* declarations for the standard command routines to those for your own commands, and changing the name of the table *standardcaproc*[ ] to something like *mycaproc*[ ], and replacing its contents with the names of your commands and their corresponding routines. Change the name of the *CONCHTABLE castandard_table* to something like *my_table*.

Edit the file *main.c*, adding an *extern* declaration for *CONCHTABLE my_table*. Change the line

*(void) caset_commands((CONCHTABLE *) NULL);*

to

(void) caset_commands(&my_table);.

Change the line

(void) caset_prompt("conch..");

to contain whatever command-prompt string you would like to see on dexter's screen.

Finally, edit *Makefile*, changing the *CFILES* list to be *main.c*, *commands.c*, and the names of the *.c* files containing your private command routines, and changing the *OFILES* list to be the names of the corresponding *.o* files. Change all occurrences of the string '*conch*' to the name you wish to give to your private version of the control program, say '*myconch*'.

Build myconch by typing the command '*make myconch*'. You can run your private version of dexter's program by typing the command '*pl -hdexter myconch*' during the downloading procedure. This program will display on dexter's screen the command prompt you have chosen, and will include in its command repertoire the commands you have listed in *mycaproc*[ ] in addition to those listed in *standardcaproc*[ ].

If users of the four-finger manipulator have occasion to make more fundamental alterations to its software in the future, it is possible that new frame types might have to be added as an aid to debugging. In order to accomplish this it is necessary to make changes in both dexter's control software and in arcopy, the program used to view archive files on the Suns.

Suppose, for instance, that we wish to add a new inner-control-related frame type *ctnew*. The first step is to add the defined-type structure declaration *CTNEW* in the appropriate header file, in this case *control.h*. The first field of this structure must be of type *FMHEAD* and must be named *fmh*.

Information specific to this frame is given in subsequent fields, subject to the constraint that the sum of the sizes in bytes of these fields must be a multiple of the value of the macro *NULLFRAME_SIZE* defined in the header file *frame.h*. Currently the null-frame size is 4 bytes.

Next, we make a type code for this new frame, defined in the header file *frame.h* by the macro *FM_CTNEW*. These type codes are listed in groups corresponding to the four-finger development directories, with each group starting at a different multiple of 10. The inner-control-related frame types are based at 60, and we add our definition to the end of this group.

We must define a bit mask in *frame.h* that will serve as a probe for the 32-element bit vector that records, for each frame type, whether it is being allocated from the main or the alternate frame pool. We call this mask *FM_CTNEWBIT*, and define it at the end of the current list of bit masks as the next power of two available.

In the file *fm_on_archive.c* in the *frame* directory we add the string '*ctnew*' to the table of strings *frametypes*[], and we add the macro *FM_CTNEWBIT* to the table of bit masks *framebits*[] in the corresponding position. The routines *fm_on_archive*( ) and *fm_off_archive*( ) allow a particular frame type to be added to or deleted from the set of frame types being allocated from the main frame pool, and also allow the contents of this set (and its complement) to be displayed on dexter's screen.

Wherever a frame of the new type is required to be allocated, the bit vector *fmpoolbits* must be masked by *FM_CTNEWBIT*. If the result is 1, the frame will be allocated from the main frame pool by a call to *fmget*( ); if the result is 0, it will be allocated from the alternate frame pool by a call to *fmaltget*( ). The code for such a call looks like (assuming that *ctnewp* is a pointer to the new frame type):

*ctnewp = (CTNEW *) fmget(FM_CTNEW, sizeof(CTNEW));*.

Most of the changes we must make to arcopy in order to accommodate the new frame type are analogous to those we made to dexter's control program. A bit mask called *CTNEWBIT* must be added to the header file *arcopy.h* in the *arcopy* directory, which will be the next power of two available in the list of these macros.

In the file *acoptions.c*, a string '*ctnew*' must be added to the list of strings *frametypes*[ ], and the macro *CTNEWBIT* must be added in the corresponding position to the table of bit masks *framebits*[ ]. This allows printing of the new frame type to be selected or suppressed in the list of options to arcopy.

The routine *acctnewprint*( ), which prints the fields of the new frame type, must be created in the file *acctnewprint.c*. This may be done by examining and imitating the existing frame-printing routines.

An *extern* declaration must be added for the routine *acctnewprint*( ) in the file *acmain.c*. In addition, a pointer to this routine must be added to the appropriate frame group's list of routine pointers (in this case, the table *ct_routines*[]).

Finally, *Makefile* in the *arcopy* directory must be edited, in order to add *acctnewprint.c* to the list of *.c* files, and to add the corresponding object file to the list of *.o* files. The new version of arcopy may now be created and installed by typing the command '*make install*' while in the *arcopy* directory. This new arcopy will be able to print out (or at least skip over) the newly-added frame type in archive files created using the new version of dexter's control program.

Two conch commands which may be of value to the advanced experimenter are the '*constant*' and '*experiment*' commands. These commands allow the run-time alteration of tables which are initialized when dexter's control software is compiled. It would ordinarily be most convenient to place these commands in a batch file to be executed during conch's initialization phase. These commands should be used with caution, as they are capable of altering the basic parameters of dexter's control software.

The '*constant*' command permits the examination or alteration of the fields of the *MCCONS* structure declared in the file *mcinit.c* in the *finger* directory on robroy. When issued with no arguments, this command displays the values of all MCCONS fields for all motor pairs. When given with a motor-pair-id argument (1, 2, 3, or 4), it displays the values of all MCCONS fields for the specified motor pair. When given with a motor-pair-id argument and an MCCONS field name, it displays the value of the specified MCCONS field for the specified motor pair. Finally, when given with a motor-pair-id argument and an additional argument of the form '*constant_type=constant_value*', the command will alter the value of the specified MCCONS field for the specified motor pair.

The '*experiment*' command serves a similar function with respect to the *EXP* structure declared in the file *grready.c* in the *conch* directory on robroy. The first argument to this command, if present, is the name of an experiment as it would be given to the '*ready*' command. A subsequent argument, if present, is or contains the name of a field of the EXP structure. The '*experiment*' command is used in a manner analogous to the '*constant*' command to display all fields or a selected field of the EXP structure for all experiments or a selected experiment, or to alter a selected field of the EXP structure for a selected experiment.

6 Debugging and performance monitoring

A number of facilities are provided to the programmer to fine tune and debug dexter's control software. The simplest of these are the '*time*' and '*gtime*' commands. The first is used to measure the time taken to carry out a conch command. It takes the name of the command to be timed as its argument, followed by the arguments of the command being timed. Following the completion of the command being timed, conch will print out how many seconds the command took to execute.

The '*gtime*' command provides a similar service at the level of the group process. If a group-id is supplied as an argument to this command, timing will be turned on for the specified group; if no group-id is given, timing will be turned on for all groups. When timing is turned on for a group, the group's conch will print a message for every command sent to that group giving the number of seconds required to execute the command. Timing is turned off for a particular group by typing the '*nogtime*' command with the group-id as argument; the '*nogtime*' command with no argument turns off timing for all groups.

More detailed performance monitoring may be obtained by using the '*profile*' command. This command initializes a buffer containing a 16-bit counter for each byte of code in dexter's control program, and causes NRTX to periodically examine the value of the CPU's program counter and increment the corresponding counter in the profile buffer. The '*noprofile*' command causes the counters to stop being incremented.

The '*save*' command causes the contents of the profile buffer to be saved to a file on robroy. This file has a name of the form */usr/tmp/pr_02.01.87_13:25:35* with the current date and time. As usual, dexter must have established an Ethernet connection with a fileserver on robroy in order to create this file.

The profiled data may be examined by using a utility on robroy called *procopy.* This program correlates the symbol tables of conch and NRTX with the counters in the saved profile buffer, and generates a list of subroutine names sorted by the amount of time dexter's CPU spent in each subroutine. Procopy reads from the profiled-data file given as argument, and writes to standard output. On-line documentation is available on procopy by typing the program name with no arguments.

Procopy reads its symbol tables by default from the files */usr/local/nrtx/remote/dexter* for NRTX and */usr/src/local/ffm/bin/conch* for conch. These can be changed by giving the desired pathnames with the *-N=* option for NRTX, or the *-C=* option for conch. Several other options may be given to procopy, details of which may be found by reading procopy's on-line help.

Note that procopy must create a symbol table for conch which has been relocated to the address at which conch will be loaded on dexter. It is therefore a good idea to check before using procopy that the macro *BASE_ADDRESS* in */usr/src/local/ffm/nsrc/procopy/Makefile* is set to the same address as was

displayed on robroy's screen during execution of the *pl* command that downloads conch to dexter. If this *Makefile* macro is changed, a new procopy will have to be made by typing the command '*make install*' in the *procopy* directory on robroy.

Rudimentary machine-language debugging capabilities are provided by the PROM monitor packaged with the Pacific Microsystems PM68D single-board computer in dexter's controller. The PROM monitor commands are given in the PROM Monitor User's Manual contained in the black three-ring binder with "Dexter" printed on the spine. This binder contains all the documentation on the Pacific Microsystems board and on NRTX.

The PROM monitor may be entered at any time by typing the 'break' key on dexter's console, in response to which the PROM monitor command prompt '>' will appear on the screen. The PROM monitor is exited by typing the monitor's 'c' command. The debugging capabilities of the PROM monitor include setting breakpoints, and examining or modifying the contents of memory or registers. All memory locations must be specified as hexadecimal numbers.

It is possible to build a control program for dexter that contains an assembly-language debugger called *ddt*, which has facilities similar to those of the UNIX debugger *adb*. This debugger has symbolic capabilities and a richer command set than the Pacific Microsystems PROM monitor, and is not restricted to use with the Pacific Microsystems board. A description of ddt's command syntax and the list of its command set are given below.

To make a version of conch with the debugger, type the command '*make ddt*' in the *conch* directory on robroy. This will produce an object file *main.o* that does not contain a dummy version of the ddt routine *mainddt*( ), and will build an executable conch containing the ddt code. Afterwards, any '*make conch*' or '*make install*' command will also produce a conch with ddt linked in.

To make a version of conch without the debugger, type the command '*make noddt*' in the *conch* directory on robroy. This will build an executable conch which does not contain the ddt code, and will also leave in the *conch* directory a version of the object file *main.o* containing a dummy version of the routine *mainddt*( ). This dummy entry point for ddt will prevent the ddt code from being linked to any conch created as the result of subsequent '*make conch*' or '*make install*' commands.

To use ddt (while running a version of conch that has the ddt code linked in), type the command '*ddt*' on dexter in response to conch's command prompt. An optional filename argument, if present, is interpreted as the absolute path name of an executable file on robroy whose symbol table is to be read by ddt. This symbol-table information permits symbol names to be used in place of addresses as command arguments. In order to access the named file, conch must be able to establish an Ethernet connection in the usual way with a fileserver running on robroy.

In order to contain valid address information, the executable file on robroy whose symbol table is read in by ddt on dexter must have been relocated by the UNIX *ld* program to the same base address it will occupy when downloaded to dexter. Such a relocated executable may be created in the *conch* directory on robroy by typing the command '*make conch.relocated*'. It is a good idea to be certain that the macro *BASE_ADDRESS* in *Makefile* is set to the same address as was displayed on robroy's screen during execution of the *pl* command that downloads conch to dexter.

The usual debugging procedure is to enter ddt and read a symbol table by means of the '*ddt*' command issued to conch, to set one or more breakpoints by means of ddt's '*esc-b*' command (see below), and then to continue execution of conch by typing an '*esc-g*' or '*esc-q*' to ddt. When a breakpoint is encountered (or a signal such as a bus error or a divide-by-zero trap), ddt's command loop will again be entered, allowing the programmer to examine or change values of variables, examine the stack, and remove or set breakpoints. Execution of conch may again be resumed by typing an '*esc-g*' or an '*esc-q*'. Note that it is not necessary to read in a symbol table in order to use ddt, but in such a case all command arguments must be hexadecimal addresses.

Additional symbol tables may read in by typing the conch '*ddt*' command with the appropriate filename argument. Only the last two symbol tables read are remembered; when a new symbol table is read in, any table read in prior to the last-read symbol table is discarded. One of the two retained symbol tables is treated as the "current" symbol table during the interpretation of command arguments.

Ordinarily, the most recently read symbol table is treated as the current one, but the other symbol table may be made current by typing the ddt '*esc-y*' command (see below). This ability to toggle between two symbol tables permits the convenient debugging of NRTX as well as conch. To read in the NRTX symbol table, the user would type the conch command '*ddt /usr/local/nrtx/remote/dexter*'.

The following description of the ddt command set is taken from the documentation provided to us along with the ddt source code by the Ultracomputer laboratory. A ddt command consists of an argument followed by a command keyword. An argument may be a numeric constant, simple expression, symbol, or register specification.

A command keyword may be either:

a single character, or

one or more escape characters (esc-)

followed by an optional single character prefix

followed by a single character.

Here is the list of command keywords:

/	Open a location, with increment.
\	Open a location, with decrement. For both / and \, the location is opened and you can modify it if you choose by typing a new value followed by a newline or another / or \. If you don't want to modify the location, just type return.
=	Print argument.
\r \n	If there is an open location (/ or \ command) close it, after changing it if argument is given.
<	Set low limit for searching. This command is only valid when followed by the > or ? commands.
>	Set high limit for searching. This command is only valid when followed by the ? command.
esc-? ?	Search for a value. Mode can be given as prefix. Prefix values are given below with the esc-t command.
esc-q	Quit ddt. Exit from ddt's command loop and continue running conch. This command may be used to continue on from a breakpoint.
esc-g	Set pc to some value and go. With no argument, this command is interchangeable with esc-q.
esc-p	Continue a certain number of times thru a breakpoint after stopping at it.
esc-esc-x esc-x	Single step a certain number of times. If more than one escape is given, then subroutines will be stepped over. If no argument is given, count is taken as one.

esc-b Set or clear a breakpoint, or print breakpoints. The optional argument determines whether breakpoint clearing or setting is to be done. If omitted, all breakpoint values are printed. If non-zero, a breakpoint is set. If zero, a breakpoint is cleared. The optional prefix determines which breakpoint will be affected. If omitted, then ddt will choose a breakpoint to set or will clear all breakpoints.

esc-r Set input and/or output radix (prefix i or o required). The default is base 16. Omitting the argument will restore the default.

esc-esc-t

esc-t Change type out mode.
The mode is given by specifying one of the following characters as the prefix:

c	character mode
h	byte mode
w	word mode
l	longword mode
i	instruction mode
s	string mode
0	default pseudo-intelligent mode

There are two separate type-out modes, depending on whether you give one or two escapes.

esc-m Set mask.
When using the ? command, the mask defines significant bit positions.

\t Save ("dot") on address stack.
An argument may be given with will be the new dot.

esc- Pop address from address stack; it then becomes dot.

esc-y Toggle between two different symbol tables. If an argument is given, it selects which symbol table: non-zero gives the oldest one, zero gives the one most recently read in.

esc-v Print a stacktrace.
You can give an argument which will limit the number of levels of stack trace. The default is 12.

esc-C Call subroutine.
Argument gives subroutine address; you will be prompted for subroutine arguments.

esc-R Print registers.
The contents of all the machine registers are displayed on the screen in the same format they are shown in when a breakpoint occurs.

NYU COMPSCI TR-282 c.1
Fehlinger, Jim
An experimenter's guide to
the four-finger
manipulator.